An Equivalence Checking Mechanism for Handling Recurrences in Array-Intensive Programs

Kunal Banerjee *

Dept. of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
kunalb@cse.iitkgp.ernet.in

1. Problem statement

Compiler optimization of array-intensive programs involves extensive application of loop transformations and arithmetic transformations. Hence, translation validation of array-intensive programs requires manipulation of sets and relations of integer points (representing array indices) bounded by constraints to account for loop transformations and simplification of arithmetic expressions to handle arithmetic transformations. A major obstacle for verification of such programs is posed by the presence of recurrences, where an element of an array gets defined in terms of some other previously defined element(s) of the same array. Recurrences lead to cycles in the data-dependence graph of the program which make dependence analyses and simplifications (through closed-form representations) of the data transformations difficult. In this work, array data-dependence graphs (ADDGs) are used to represent both the original and the optimized version of the program and a validation scheme is proposed where the cycles in the ADDGs are isolated from the acyclic portions and treated separately. Thus, this work provides a unified equivalence checking framework to handle loop and arithmetic transformations along with recurrences – this combination of features had not been achieved by a single verification technique earlier.

2. Literature survey

Loop transformations together with arithmetic transformations are applied extensively in the domain of multimedia and signal processing applications to obtain better performance in terms of energy, area and/or execution time. The work reported in [1], for example, applies loop fusion and loop tiling to several nested loops and parallelizes the resulting code across different processors for multimedia applications. Minimization of the total energy while satisfying the performance requirements for applications with multi-dimensional nested loops was targeted in [2]. Arithmetic code transformations have been successfully employed in minimizing critical path lengths [6]. Importantly, loop transformation and arithmetic transformation techniques are applied dynamically since application of one may create scope of application of several other techniques. In all these cases, it is crucial to ensure that the intended behaviour of the program has not been altered wrongly during transformation.

An approach for verifying array-intensive programs can be to use off-the-shelf SMT solvers or theorem provers since the equivalence between two programs can be modeled with a formula such that the validity of the formula implies the equivalence [4]. Although SMT solvers and theorem provers can efficiently handle linear arithmetic, they are not equally suitable for handling non-linear arithmetic which is often encountered in array-intensive programs; hence, these tools are found to be inadequate for establishing equivalence of such programs [4]. The works reported in [8, 9] consider a restricted class of programs which must have static control-flows, uniform recurrences, valid schedules, affine indices and bounds and single assignment forms. In [8, 9], the original and the transformed behaviours are modeled as ADDGs and the correctness of the loop transformations is established by showing the equivalence between the two ADDGs. These works are capable of handling a wide variety of loop transformation techniques without taking any information from the synthesis tools. The method proposed in [10, 11] extends the ADDG model to a dependence graph model to handle recurrences, both uniform and non-uniform, along with associative and commutative operations. All the above methods, however, fail if the transformed behaviour is obtained from the original behaviour by application of arithmetic transformations such as, distributive transformations, arithmetic expression simplification, common sub-expression elimination, constant unfolding, etc., along with loop transformations. The work reported in [3, 5] furnishes an ADDG based method which compares ADDGs at slice-level rather than path-level as performed in [8] and employs a normalization technique [7] for the arithmetic expressions to verify a wide variety of loop transformations and a wide range of arithmetic transformations applied together in array-intensive programs. However, it cannot verify programs involving recurrences because recurrences lead to cycles in the ADDGs which are, otherwise, directed acyclic graphs. The presence of cycles makes the existing data-dependence analysis and simplification (through closed-form representations) of the data transformations in ADDGs inapplicable.

3. Extension of the equivalence checking scheme to handle recurrences

This work provides a unified equivalence checking framework based on ADDGs to handle loop and arithmetic transformations along with recurrences. The validation scheme proposed here isolates the cycles in the ADDGs from the acyclic portions and treats them separately; a cycle (arising from a recurrence) in the original ADDG is compared with its corresponding cycle in the transformed ADDG in isolation while the acyclic portions are compared using the conventional technique of [5]. This technique, however, has the following restrictions: (i) since an array element in a recurrence gets defined in terms of element(s) of the same array, the loop iterators of both the ADDGs must proceed in an identical fashion, i.e., both should increment or decrement identically, and (ii) a recurrence in the original program can be divided (merged) in the transformed version by applying loop fission (loop fusion); in this work, we target only those programs whose recurrences undergo no such loop fusion/fission.

The equivalence checking technique for handling recurrences is illustrated with an example taken from [11]. Let us consider the pair of equivalent programs involving recurrences as shown

* The work presented here was supported by TCS Research Fellowship. The author acknowledges C Mandal and D Sarkar, Dept of CSE, IIT Kharagpur, for their technical participation in this work.

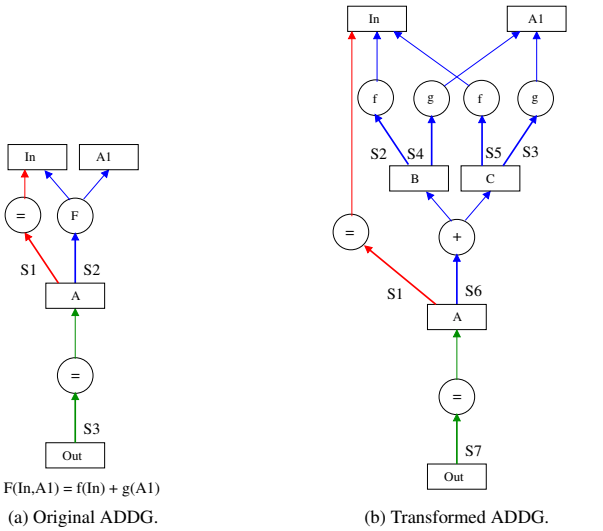
```
S1: A[0] = In[0];
for (i = 1; i < N; ++i) {
 S2: A[i] = f(In[i]) +
            g(A[i-1]);
}
S3: Out = A[N-1];
```

(a) Original program.

```
S1: A[0] = In[0];
for (i = 1; i < N; ++i) {
 if ( i%2 == 0 ) {
  S2: B[i] = f(In[i]);
  S3: C[i] = g(A[i-1]);
 } else {
  S4: B[i] = g(A[i-1]);
  S5: C[i] = f(In[i]);
 }
 S6: A[i] = B[i] + C[i];
}
S7: Out = A[N-1];
```

(b) Transformed program.

Figure 1: An example of two programs containing recurrences.



(a) Original ADDG. (b) Transformed ADDG.

Figure 2: ADDGs for the programs given in Fig. 1.

in Fig. 1. To remove the cycle corresponding to Fig. 1(a), we introduce a *new* (not already defined) array variable `A1` to replace the array `A` occurring on the right hand side of the statement `S2`. Similarly, we replace `A` with the same array variable `A1` in the statements `S3` and `S4` of the program given in Fig. 1(b) to remove the cycles in its corresponding ADDG. The modified ADDGs, without the cycles, are shown in Fig. 2. Note that application of loop transformations, such as loop reversal in the present example, will result in a non-valid (and thus non-equivalent) program – the equivalence checking procedure will not be able to detect this mistake once the new array element `A1` has been introduced. Hence, we need the first restriction mentioned above. The equivalence of the two ADDGs given in Fig. 2(a) and Fig. 2(b) can be established in the following manner. First of all, scalar variables such as `Out` are treated as array variables of unit dimension, i.e., `Out` is considered as `Out[0]`, also note that the iteration domains for statements such as `S1` and `S3` in Fig. 1(a) are trivial. Showing equivalence of the acyclic subgraphs of the ADDGs, denoted by red and green arrows in Fig. 2, is straightforward by the method of [5]. Now, let us concentrate on the blue portions of the two ADDGs. Here, we establish the data transformation of `A` in terms of `In` and `A1`; since the nodes corresponding to newly introduced array variables such as, `A1`, will not have any outgoing edges, they are treated as input arrays while checking equivalence of the recurrences.

The mapping representing how the members of the output array out are related to those of the input arrays $in1$ and $in2$ is denoted by $M_{out,\{in1,in2\}}$ while the corresponding data transformation is denoted by $r_{out,\{in1,in2\}}$; the detailed formalism can be found in [5]. For the ADDG shown in Fig. 2(a), $M_{A,In} = \{[i] \rightarrow [i] \mid 1 \leq i < N\}$, $M_{A,A1} = \{[i] \rightarrow [i-1] \mid 1 \leq i < N\}$ and $r_{A,\{In,A1\}} = f(In) + g(A1)$. For the ADDG shown in Fig. 2(b), $M_{B,In} = \{[i] \rightarrow [i] \mid \exists k \in \mathbb{Z}(i = 2 \times k), 1 \leq i < N\}$, $M_{C,A1} = \{[i] \rightarrow [i-1] \mid \exists k \in \mathbb{Z}(i = 2 \times k), 1 \leq i < N\}$, $M_{B,A1} = \{[i] \rightarrow [i-1] \mid \exists k \in \mathbb{Z}(i = 2 \times k + 1), 1 \leq i < N\}$, $M_{C,In} = \{[i] \rightarrow [i] \mid \exists k \in \mathbb{Z}(i = 2 \times k + 1), 1 \leq i < N\}$, $M_{A,B} = \{[i] \rightarrow [i] \mid 1 \leq i < N\}$, $M_{A,C} = \{[i] \rightarrow [i] \mid 1 \leq i < N\}$. Now, we find $r^{(1)}_{A,\{In,A1\}} = f(In) + g(A1)$ for the domain $\{[i] \mid \exists k \in \mathbb{Z}(i = 2 \times k), 1 \leq i < N\}$ and $r^{(2)}_{A,\{In,A1\}} = f(In) + g(A1)$ for the domain $\{[i] \mid \exists k \in \mathbb{Z}(i = 2 \times k + 1), 1 \leq i < N\}$; note that our normalization technique knows that $+$ is a commutative operation. Since the data transformation is same in both the domains, we can combine them and thus we arrive at the same mappings and data transformation as that of Fig. 2(a). Therefore, we conclude that the ADDGs of Fig. 2 and in turn, the programs of Fig. 1, are equivalent.

4. Conclusion and future work

This work, for the first time, provides a unified equivalence checking framework for handling loop transformations and arithmetic transformations along with recurrences. We also aim to handle sophisticated recurrences that cannot be verified using the method of [11] by following a template matching procedure (which could not be covered here due to page limitation). The method described here has been implemented, other enhancements are being incorporated.

References

[1] Y. Bouchebaba, B. Girodias, G. Nicolescu, E. M. Aboulhamid, B. Lavigueur, and P. G. Paulin. MPSoC memory optimization using program transformation. *ACM Trans. Design Autom. Electr. Syst.*, 12(4), 2007.

[2] I. Kadayif, M. T. Kandemir, G. Chen, O. Ozturk, M. Karaköy, and U. Sezer. Optimizing array-intensive applications for on-chip multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 16(5):396–411, 2005.

[3] C. Karfa, K. Banerjee, D. Sarkar, and C. Mandal. Equivalence checking of array-intensive programs. In *ISVLSI*, pages 156–161, 2011.

[4] C. Karfa, K. Banerjee, D. Sarkar, and C. Mandal. Experimentation with SMT solvers and theorem provers for verification of loop and arithmetic transformations. In *I-CARE*, pages 3:1–3:4, 2013.

[5] C. Karfa, K. Banerjee, D. Sarkar, and C. Mandal. Verification of loop and arithmetic transformations of array-intensive behaviours. *IEEE Trans. on CAD of ICS*, 32(11):1787–1800, 2013.

[6] B. Landwehr and P. Marwedel. A new optimization technique for improving resource exploitation and critical path minimization. In *ISSS*, pages 65–72, 1997.

[7] D. Sarkar and S. De Sarkar. A theorem prover for verifying iterative programs over integers. *IEEE Trans Software. Engg.*, 15(12):1550–1566, 1989.

[8] K. C. Shashidhar. *Efficient Automatic Verification of Loop and Dataflow Transformations by Functional Equivalence Checking*. PhD thesis, Katholieke Universiteit Leuven, 2008.

[9] K. C. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens. Functional equivalence checking for verification of algebraic transformations on array-intensive source code. In *DATE*, pages 1310–1315, 2005.

[10] S. Verdoolaege, G. Janssens, and M. Bruynooghe. Equivalence checking of static affine programs using widening to handle recurrences. In *CAV*, pages 599–613, 2009.

[11] S. Verdoolaege, G. Janssens, and M. Bruynooghe. Equivalence checking of static affine programs using widening to handle recurrences. *ACM Trans. Program. Lang. Syst.*, 34(3), 2012.